Part 9

Functions

Functions

- Programming languages let you define functions

```
def add(x,y):
    return x+y

def countdown(n):
    while n > 0:
        print("T-minus",n)
        n -= 1
    print("Boom!")
```

- Two problems:
    - Scoping of identifiers
    - Runtime implementation

Function Scoping

- Most languages use lexical scoping
- Pertains to visibility of identifiers

```
a = 13
def foo():
    b = 42
    print(a,b)          # a,b are visible

def bar():
    c = 13
    print(a,b)          # a,c are visible
                        # b is not visible
```

- Identifiers defined in enclosing source code context of a particular statement are visible

Python Scoping

- Python uses two-level scoping
  - Global scope (module-level)
  - Local scope (function bodies)

```
a = 13                 # Global
def foo():
    b = 42             # Local
    print(a,b)
```

Block Scoping

- Some languages use block scoping (e.g., C)

```
int a = 1;              / Global
int foo() {
    int n = 0;          / Local. Visible in entire func
    while (n < 10) {
        int x = 2;      / Block. Visible only in 'while'
        ...
    }
    printf("%d\n",x); / Error. x not defined
}
```

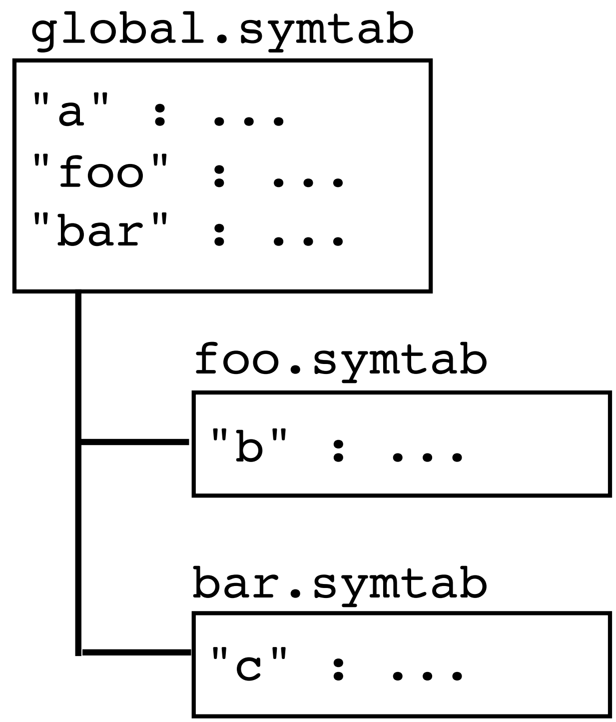
- Not in Python though...

Scope Implementation

- In the compiler: nested tables

```
a = 13
def foo():
    b = 42
    print(a,b)

def bar():
    c = 13
    print(a,b)
```



- Symbol table lookup checks all parents
- The nesting is by syntactic/lexical structure

Function Runtime

- Each invocation of a function creates a new environment of local variables
- Known as an activation frame (or record)
- Activation frames make up the call stack

Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```

Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```

foo

```
a : 1
b : 2
c : 3
```

Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```

foo	a : 1 b : 2 c : 3
bar	x : 3 y : 6

Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```

foo	a : 1 b : 2 c : 3
bar	x : 3 y : 6
spam	z : 6

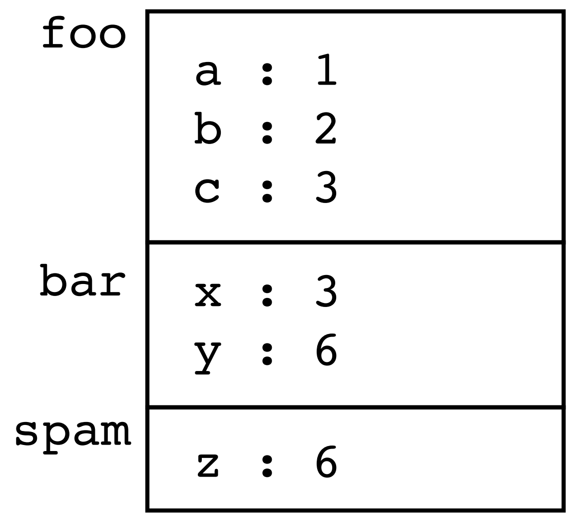
Activation Frames

```
def foo(a,b):
    c = a+b
    bar(c)

def bar(x):
    y = 2*x
    spam(y)

def spam(z):
    return 10*z

foo(1,2)
```



Note: Frames are NOT related to scoping of variables (functions don't see the variables defined inside other functions).

Activation Frames

- You see frames in tracebacks

```
File "expr.py", line 20, in <module>
  exprcheck.check_program(program)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 410, in check_program
  checker.visit(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 163, in visit_Program
  self.visit(node.statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
  self.visit(item)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 350, in visit_FuncDeclar
  self.visit(node.statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
  self.visit(item)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprcheck.py", line 303, in visit_IfStatemen
  self.visit(node.if_statements)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 238, in visit
  return visitor(node)
File "/Users/beazley/Desktop/Compiler/compilers/exprast.py", line 253, in generic_visit
  self.visit(item)
```

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)
```

(caller)

```
def foo(x,y):
    z = x + y
    return z
```

(callee)

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

creates →

```
x : 1
y : 2
return : None
```

Caller is responsible for creating a new frame and populating it with input arguments.

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

creates →

```
x : 1
y : 2
return : None
```

<u>Semantic Issue: What does the frame contain?</u>

Copies of the arguments? (Pass by value)

Pointers to the arguments? (Pass by reference)

Depends on the language

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

`x : 1`
`y : 2`
`return : None`

`Return PC`

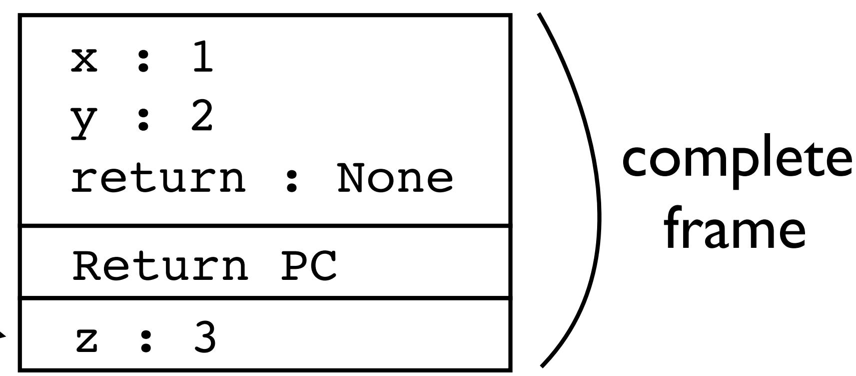
Return address (PC) recorded in the frame (so you can get back to the caller upon return)

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```



Local variables get added to the frame by the callee

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

Return result placed in frame

```
x : 1
y : 2
return : 3
----------
Return PC
----------
z : 3
```

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

return

```
x : 1
y : 2
return : 3
```

callee destroys its part
of the frame on return

Frame Management

- Management of Activation Frames is managed by both the caller and callee

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

caller destroys remaining frame on assignment of result

Frame Management

- Implementation Detail : Frame often organized as an array of numeric "slots"

```
result = foo(1,2)

def foo(x,y):
    z = x + y
    return z
```

Slot	Contents
0	x : 1
1	y : 2
2	return : None
3	Return PC
4	z : 3

complete frame

- Slot numbers used in low-level instructions
- Determined at compile-time

Frame Example

- Python Disassembly

```
def foo(x,y):
    z = x + y
    return z

>>> import dis
>>> dis.dis(foo)
  2           0 LOAD_FAST                0 (x)
              3 LOAD_FAST                1 (y)
              6 BINARY_ADD
              7 STORE_FAST               2 (z)

  3          10 LOAD_FAST                2 (z)
             13 RETURN_VALUE
>>>
```

numbers refer to "slots" in the activation frame

ABIs

- Application Binary Interface
- A precise specification of function/procedure call semantics related to activation frames
- Language agnostic
- Critical part of creating programming libraries, DLLs, modules, etc.
- Different than an API (higher level)

Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):
    ...
    return bar(a-1)

def bar(a):
    ...
    return result

foo(1)
```

```
foo  a : 1
```

compiler detects that no more statements follow

Tail Call Optimization

- Sometimes the compiler can eliminate frames

```
def foo(a):
    ...
    return bar(a-1)

def bar(a):
    ...
    return result

foo(1)
```

bar

a : 0

compiler reuses the same stack frame and just jumps to the next procedure (goto)

- Note: Python does <u>not</u> do this (although people often wish that it did)

Closures

- Nested functions are "interesting"

```
def add(x):
    def f(y):
        return x + y
    return f
```

- Example:

```
>>> a = add(2)
>>> a(3)
5
>>>
```

- The "x" variable must live someplace
- It does <u>not</u> exist on the stack.

Closures

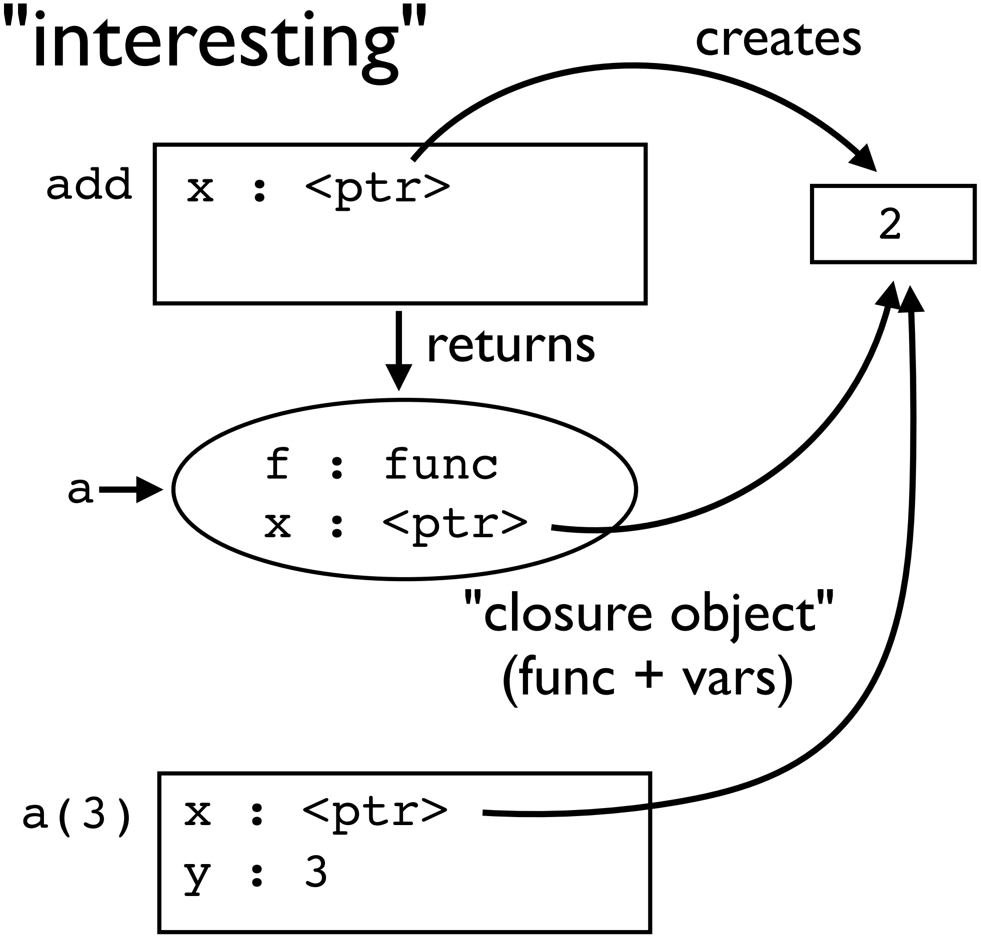
- Nested functions are "interesting"

```
def add(x):
    def f(y):
        return x + y
    return f
```

- Example:

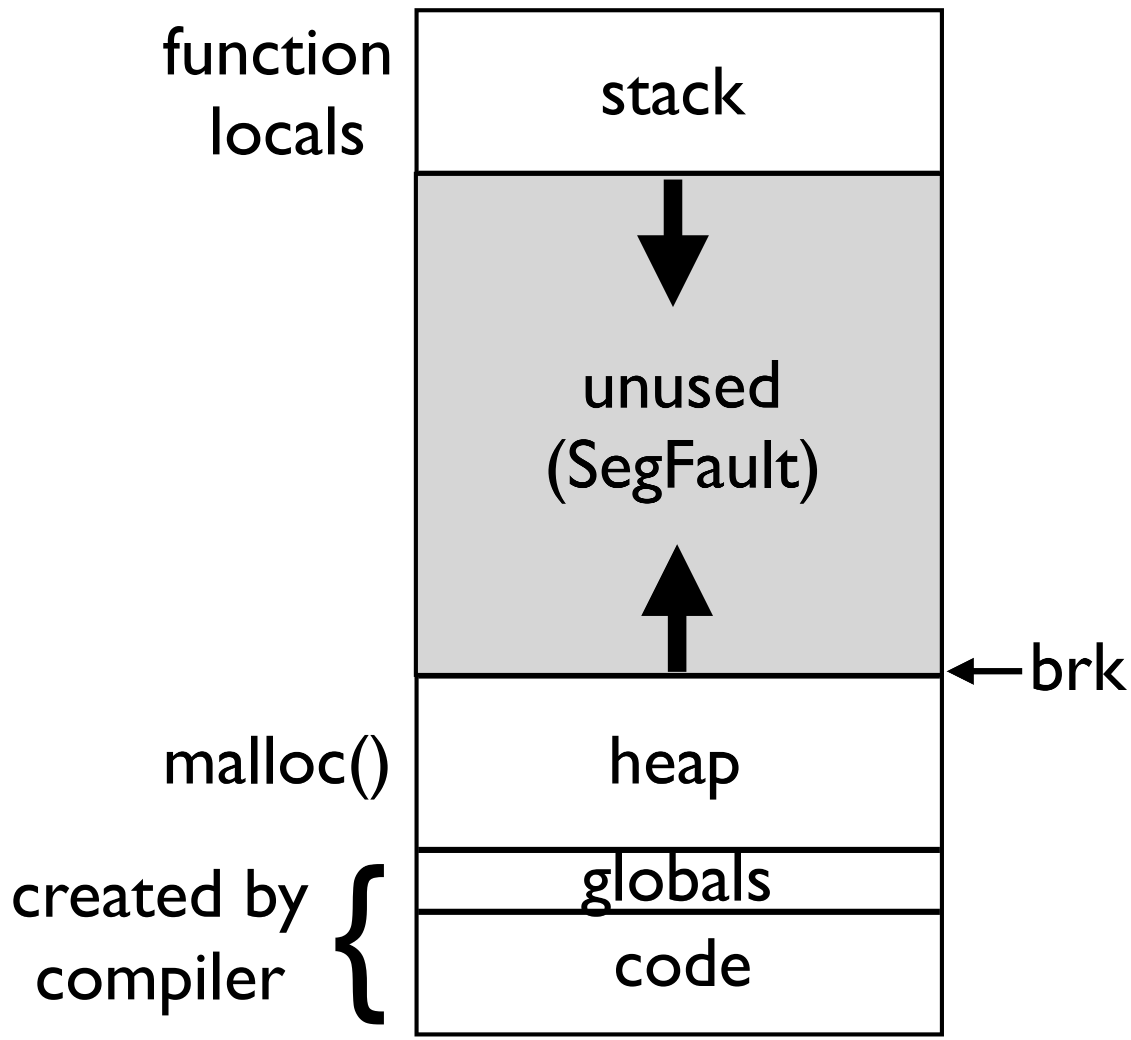
```
>>> a = add(2)
>>> a(3)
5
>>>
```

- Indirect reference to a value stored "off stack"



Memory Management

- Runtime memory layout
- Managed by the code emitted from the compiler and operating system.
- Related: Garbage Collection



Program Startup

- Most programs have an entry point
- Often called main()
- Must be written by the user

Program Startup

- Compiler generates a hidden startup/ initialization function that calls main()

```
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    ...
    return main();
}
```

- Primary purpose is to initialize globals

Program Startup

- Initialization example:

```
var x int = v1;
var y int = v2;
...
func main() int {
    // Written by the programmer
    ...
    return 0;
}

func __start() int {
    // Initialization (created by compiler)
    x = v1;     // Setting of global variables
    y = v2;
    return main();
}
```

Project

- Modify your compiler to support functions
- Requires modifications to most parts
- Will be a good review of everything!